Experiment No.: 01

Aim: Write a Python program to understand SHA and Cryptography in Blockchain, Merkle root Tree Hash.

Tasks Performed:

1. **Hash Generation using SHA-256:**
   - Developed a Python program to compute a SHA-256 hash for any given input string using the hashlib library.
2. **Target Hash Generation with Nonce:**
   - Created a program to generate a hash code by concatenating a user input string and a nonce value to simulate the mining process.
3. **Proof-of-Work Puzzle Solving:**
   - Implemented a program to find the nonce that, when combined with a given input string, produces a hash starting with a specified number of leading zeros.
4. **Merkle Tree Construction:**
   - Built a Merkle Tree from a list of transactions by recursively hashing pairs of transaction hashes, doubling up last nodes if needed, and generated the Merkle Root hash for blockchain transaction integrity.

Theory:

Cryptography in Blockchain

Cryptography is a method of securing data from unauthorized access. In the blockchain, cryptography is used to secure transactions taking place between two nodes in a blockchain network. In a blockchain there are two main concepts cryptography and hashing. Cryptography is used to encrypt messages in a P2P network and hashing is used to secure the block information and the link blocks in a blockchain. Cryptography primarily focuses on ensuring the security of participants, transactions, and safeguards against double-spending. It helps in securing different transactions on the blockchain network. It ensures that only the individuals for whom the transaction data is intended can obtain, read and process the transaction.

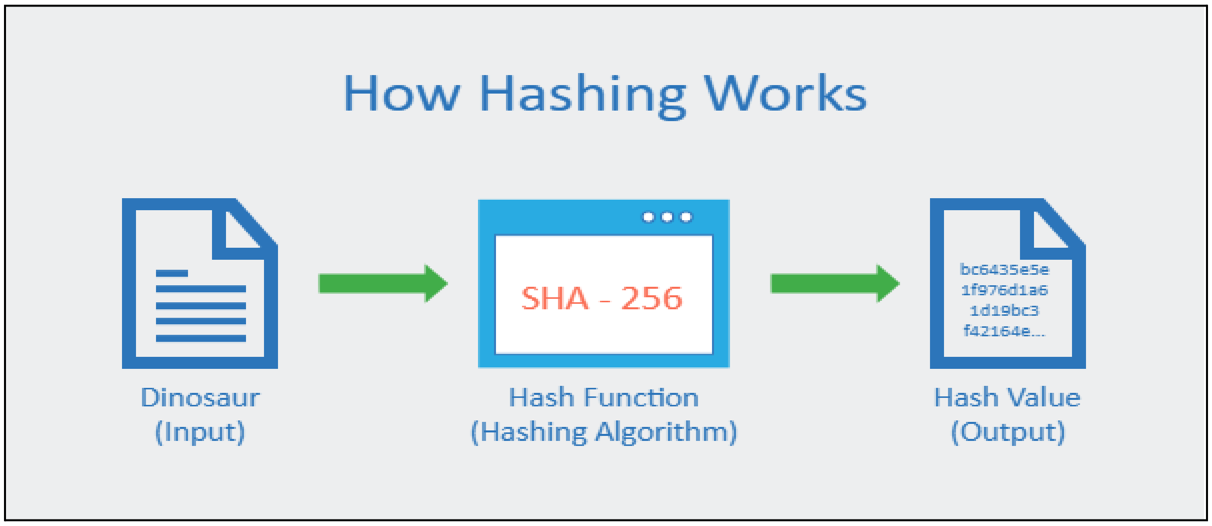
Role of Cryptography in Blockchain

Blockchain is developed with a range of different cryptography concepts. The development of cryptography technology promotes restrictions for the further development of blockchain.

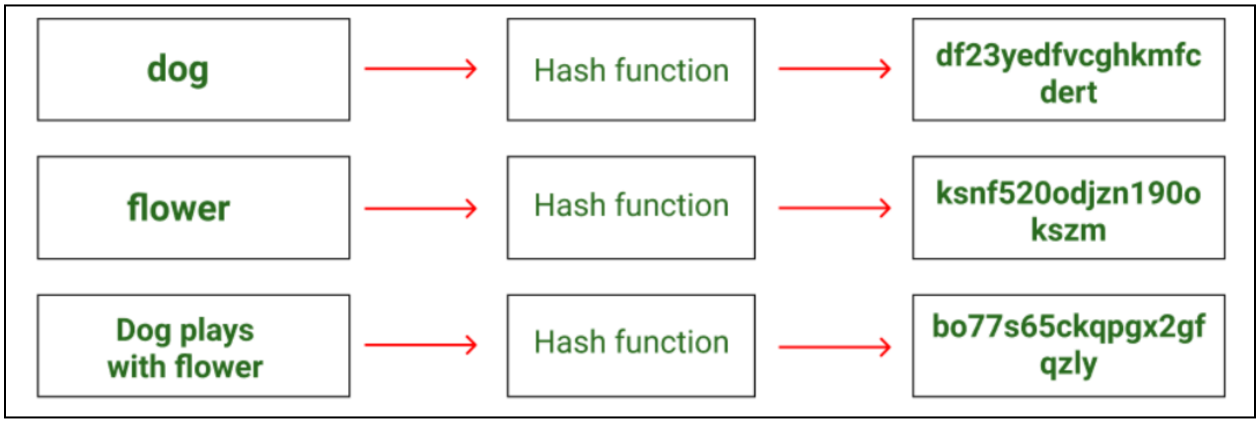
- In the blockchain, cryptography is mainly used to protect user privacy and transaction information and ensure data consistency.
- The core technologies of cryptography include symmetric encryption and asymmetric encryption.
- Asymmetric cryptography uses digital signatures for verification purposes, every transaction recorded to the block is signed by the sender by digital signature and ensures that the data is not corrupted.
- Cryptography plays a key role in keeping the public network secure, so making it fit to maintain the integrity and security of blockchain.

Cryptography Hash Function in Blockchain

One of the most notable uses of cryptography is cryptographic hashing. Hashing enables immutability in the blockchain. The encryption in cryptographic hashing does not involve any use of keys. When a transaction is verified hash algorithm adds the hash to the block, and a new unique hash is added to the block from the original transaction. Hashing continues to combine or make new hashes, but the original footprint is still accessible. The single combined hash is called the root hash. Hash Function helps in linking the block as well as maintaining the integrity of data inside the block and any alteration in the block data leads to a break of the blockchain. Some commonly used hashed function is MD5 and SHA-1.



A cryptographic hash is a function that outputs a fixed-size digest for a variable-length input. A hash function is an important cryptographic primitive and extensively used in blockchain. For example, SHA-256 is a hash function in which for any variable-bit length input, the output is always going to be a 256-bit hash.



- From the above picture, it is clear that even the slightest change in an alphabet in the input sentence can drastically change the hash obtained. Therefore hashes can be used to verify integrity.
- Consider there is a text file with important data. Pass the contents of the text file into a hash function and then store the hash in the phone. A hacker manages to open the text file and changes the data.
- Now when you open the file again, you can compute the hash again and compare this hash with the one stored previously on the phone.
- It will be clearly evident that the two hashes do not match and hence the file has been tampered with.

Properties of Cryptographic Hash:

- For a particular message hash function does not change.
- Every minor change in data will result in a change in a major change in the hash value.
- The input value cannot be guessed from the output hash function.
- They are fast and efficient as they largely rely on bitwise operations.

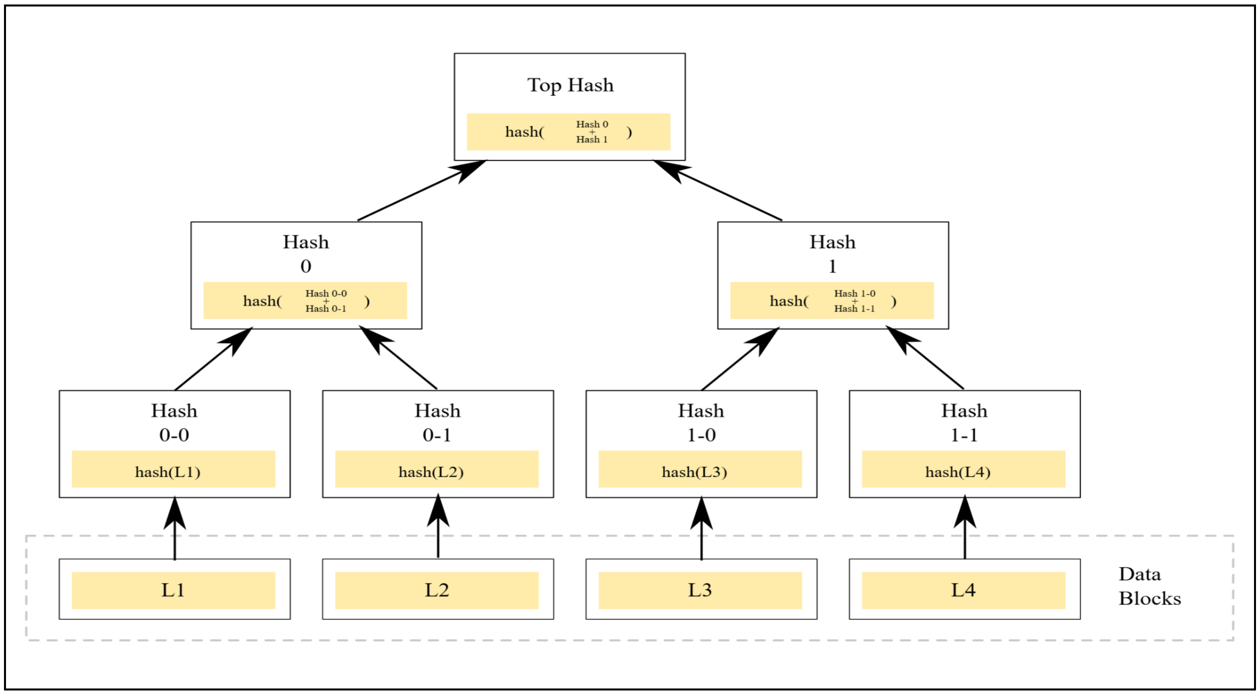
Benefits of Hash function in Blockchain:

- Reduce the bandwidth of the transaction.
- Prevent the modification in the data block.
- Make verification of the transaction easier.
- Use of Cryptographic Hash Functions

As the blockchain is also public to everyone it is important to secure data in the blockchain and keeps the data of the user safe from malicious hands. So, this can be achieved easily by cryptography. When the transaction is verified through a hash algorithm, it is added to the blockchain, and as the transaction becomes confirmed it is added to the network making a chain of blocks. Cryptography uses mathematical codes, it ensures the users to whom the data is intended can obtain it for reading and processing the transaction.

Merkle Tree

Merkle tree also known as hash tree is a data structure used for data verification and synchronization. It is a tree data structure where each non-leaf node is a hash of it's child nodes. All the leaf nodes are at the same depth and are as far left as possible. It maintains data integrity and uses hash functions for this purpose.

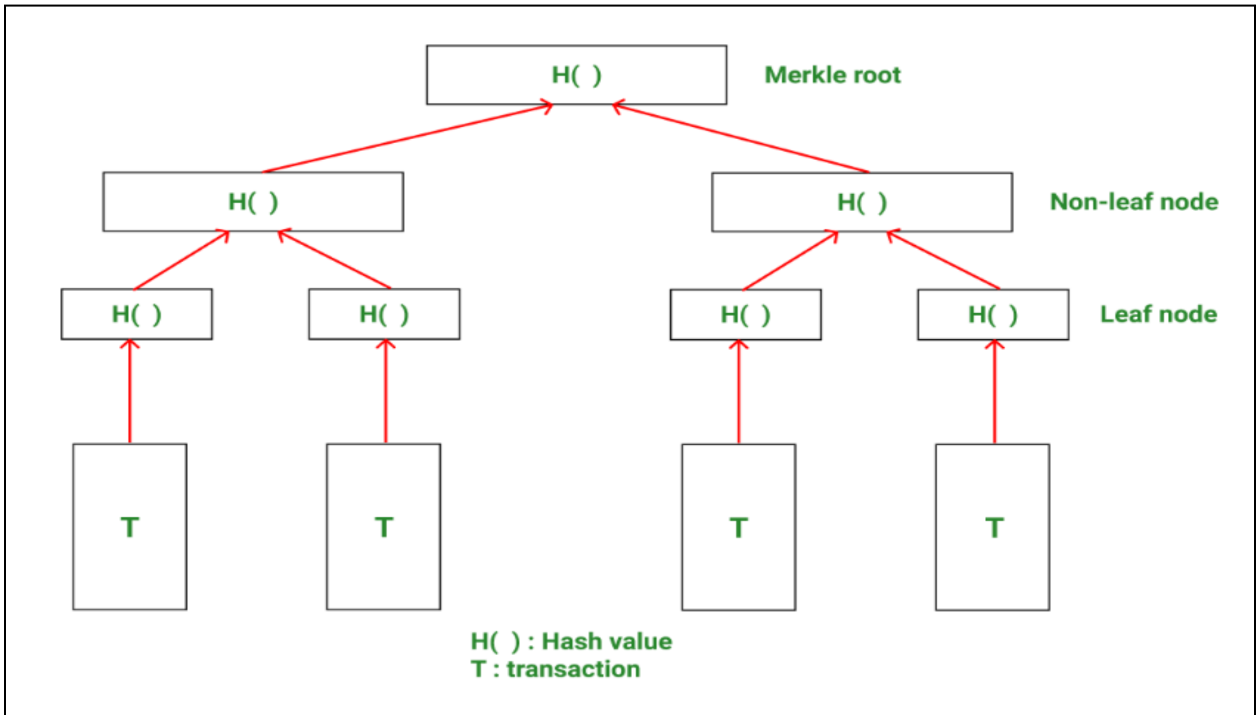


This is a binary merkel tree, the top hash is a hash of the entire tree. This structure of the tree allows efficient mapping of huge data and small changes made to the data can be easily identified. If we want to know where data change has occurred then we can check if data is consistent with root hash and we will not have to traverse the whole structure but only a small part of the structure. The root hash is used as the fingerprint for the entire data.

Applications of Markle Tree :

- Merkle trees are useful in distributed systems where same data should exist in multiple places.
- Merkle trees can be used to check inconsistencies.
- Apache Cassandra uses Merkle trees to detect inconsistencies between replicas of entire databases.
- It is used in bitcoin and blockchain.

Merkle Tree Structure



Structure of Merkle tree

1. A blockchain can potentially have thousands of blocks with thousands of transactions in each block. Therefore, memory space and computing power are two main challenges.
2. It would be optimal to use as little data as possible for verifying transactions, which can reduce CPU processing and provide better security, and this is exactly what Merkle trees offer.
3. In a Merkle tree, transactions are grouped into pairs. The hash is computed for each pair and this is stored in the parent node. Now the parent nodes are grouped into pairs and

their hash is stored one level up in the tree. This continues till the root of the tree. The different types of nodes in a Merkle tree are:

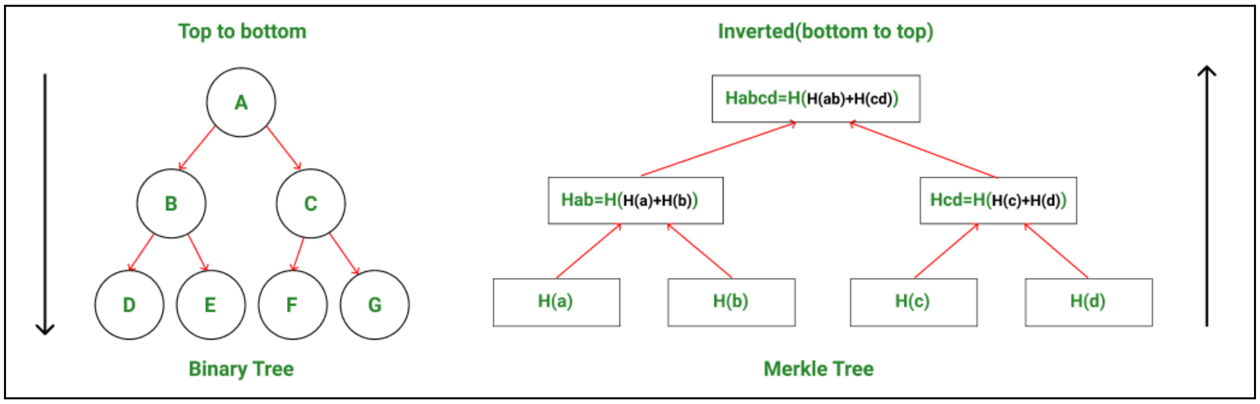
   a. **Root node:** The root of the Merkle tree is known as the Merkle root and this Merkle root is stored in the header of the block.
   b. **Leaf node:** The leaf nodes contain the hash values of transaction data. Each transaction in the block has its data hashed and then this hash value (also known as transaction ID) is stored in leaf nodes.
   c. **Non-leaf node:** The non-leaf nodes contain the hash value of their respective children. These are also called intermediate nodes because they contain the intermediate hash values and the hash process continues till the root of the tree.

4. Bitcoin uses the SHA-256 hash function to hash transaction data continuously till the Merkle root is obtained.
5. Further, a Merkle tree is binary in nature. This means that the number of leaf nodes needs to be even for the Merkle tree to be constructed properly. In case there is an odd number of leaf nodes, the tree duplicates the last hash and makes the number of leaf nodes even.

<u>How Do Merkle Trees Work?</u>

A Merkle tree is constructed from the leaf nodes level all the way up to the Merkle root level by grouping nodes in pairs and calculating the hash of each pair of nodes in that particular level. This hash value is propagated to the next level. This is a bottom-to-up type of construction where the hash values are flowing from down to up direction.

Hence, by comparing the Merkle tree structure to a regular binary tree data structure, one can observe that Merkle trees are actually inverted down.



Example: Consider a block having 4 transactions- T1, T2, T3, T4. These four transactions have to be stored in the Merkle tree and this is done by the following steps-

Step 1: The hash of each transaction is computed.

H1 = Hash(T1).

Step 2: The hashes computed are stored in leaf nodes of the Merkle tree.

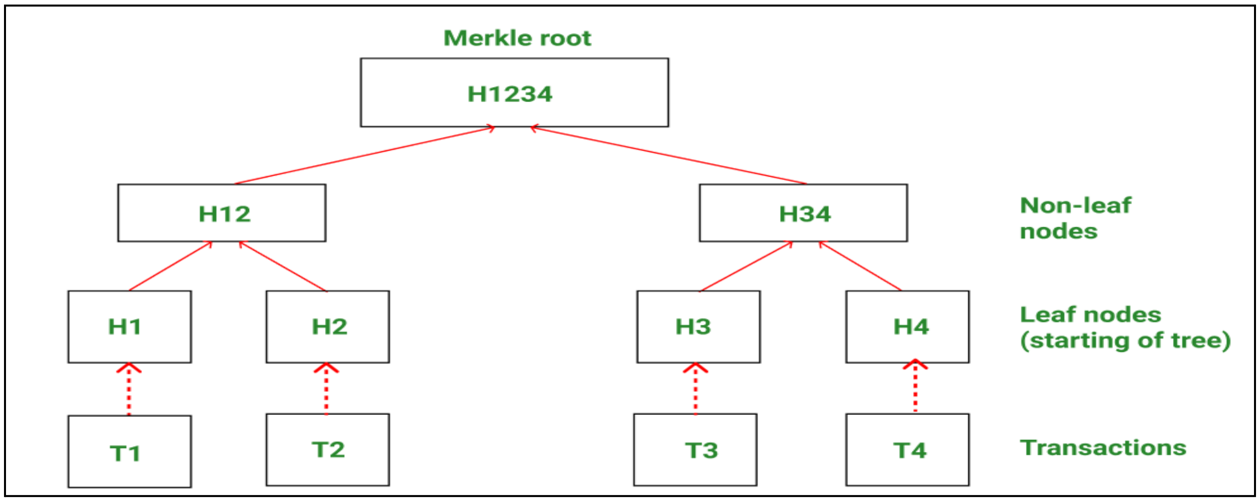
Step 3: Now non-leaf nodes will be formed. In order to form these nodes, leaf nodes will be paired together from left to right, and the hash of these pairs will be calculated. Firstly hash of H1 and H2 will be computed to form H12. Similarly, H34 is computed. Values H12 and H34 are parent nodes of H1, H2, and H3, H4 respectively. These are non-leaf nodes.

H12 = Hash(H1 + H2)

H34 = Hash(H3 + H4)

Step 4: Finally H1234 is computed by pairing H12 and H34. H1234 is the only hash remaining. This means we have reached the root node and therefore H1234 is the Merkle root.

H1234 = Hash(H12 + H34)



Merkle tree works by hashing child nodes again and again till only one hash remains.

- In order to check whether the transaction has tampered with the tree, there is only a need to remember the root of the tree.

- One can access the transactions by traversing through the hash pointers and if any content has been changed in the transaction, this will reflect on the hash stored in the parent node, which in turn would affect the hash in the upper-level node and so on until the root is reached.
- Hence the root of the Merkle tree has also changed. So Merkle root which is stored in the block header makes transactions tamper-proof and validates the integrity of data.
- With the help of the Merkle root, the Merkle tree helps in eliminating duplicate or false transactions in a block.
- It generates a digital fingerprint of all transactions in a block and the Merkle root in the header is further protected by the hash of the block header stored in the next block.

Why Merkle Trees are Important For Blockchain?

- In a centralized network, data can be accessed from one single copy. This means that nodes do not have to take the responsibility of storing their own copies of data and data can be retrieved quickly.
- However, the situation is not so simple in a distributed system.
- Let us consider a scenario where blockchain does not have Merkle trees. In this case, every node in the network will have to keep a record of every single transaction that has occurred because there is no central copy of the information.
- This means that a huge amount of information will have to be stored on every node and every node will have its own copy of the ledger. If a node wants to validate a past transaction, requests will have to be sent to all nodes, requesting their copy of the ledger. Then the user will have to compare its own copy with the copies obtained from several nodes.
- Any mismatch could compromise the security of the blockchain. Further on, such verification requests will require huge amounts of data to be sent over the network, and the computer performing this verification will need a lot of processing power for comparing different versions of ledgers.
- Without the Merkle tree, the data itself has to be transferred all over the network for verification.

- Merkle trees allow comparison and verification of transactions with viable computational power and bandwidth. Only a small amount of information needs to be sent, hence compensating for the huge volumes of ledger data that had to be exchanged previously.
- Merkle trees use a one-way hash function extensively and this hashing separates the proof of data from data itself

Advantages of Merkle Tree

Efficient verification: Merkle trees offer efficient verification of integrity and validity of data and significantly reduce the amount of memory required for verification. The proof of verification does not require a huge amount of data to be transmitted across the blockchain network. Enable trustless transfer of cryptocurrency in the peer-to-peer, distributed system by the quick verification of transactions.

No delay: There is no delay in the transfer of data across the network. Merkle trees are extensively used in computations that maintain the functioning of cryptocurrencies.

Less disk space: Merkle trees occupy less disk space when compared to other data structures.

Unaltered transfer of data: Merkle root helps in making sure that the blocks sent across the network are whole and unaltered.

Tampering Detection: Merkle tree gives an amazing advantage to miners to check whether any transactions have been tampered with.

Since the transactions are stored in a Merkle tree which stores the hash of each node in the upper parent node, any changes in the details of the transaction such as the amount to be debited or the address to whom the payment must be made, then the change will propagate to the hashes in upper levels and finally to the Merkle root.

The miner can compare the Merkle root in the header with the Merkle root stored in the data part of a block and can easily detect this tampering.

Time Complexity: Merkle tree is the best solution if a comparison is done between the time complexity of searching a transaction in a block as a Merkle tree and another block that has transactions arranged in a linked list, then-

Merkle Tree search: O(logn), where n is the number of transactions in a block.

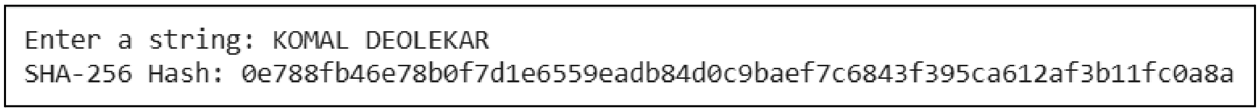
Linked List search: O(n), where n is the number of transactions in a block.

Programs And Outputs:

Program #1: Hash Generation Using hashlib Library

```python
import hashlib

def create_hash(string):
    # Create a hash object using SHA-256 algorithm
    hash_object = hashlib.sha256()
    # Convert the string to bytes and update the hash object
    hash_object.update(string.encode('utf-8'))
    # Get the hexadecimal representation of the hash
    hash_string = hash_object.hexdigest()
    # Return the hash string
    return hash_string

# Example usage
input_string = input("Enter a string: ")
hash_result = create_hash(input_string)
print("SHA-256 Hash:", hash_result)
```

Output :

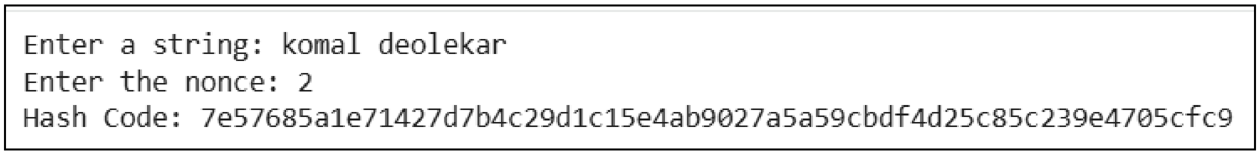


```
Enter a string: KOMAL DEOLEKAR
SHA-256 Hash: 0e788fb46e78b0f7d1e6559eadb84d0c9baef7c6843f395ca612af3b11fc0a8a
```

Program #2: Generating Target Hash with Input String and Nonce

```python
import hashlib

def generate_hash(input_string, nonce):
    # Concatenate the string and nonce
    hash_string = input_string + nonce

    # Calculate the hash using SHA-256
    hash_object = hashlib.sha256(hash_string.encode('utf-8'))
    # Convert the hash into hexadecimal format
    hash_code = hash_object.hexdigest()

    # Return the hash code
    return hash_code

# Get user input
input_string = input("Enter a string: ")
nonce = input("Enter the nonce: ")

# Call the function to generate hash
result = generate_hash(input_string, nonce)

# Print the hash code
print("Hash Code:", result)
```

Output :

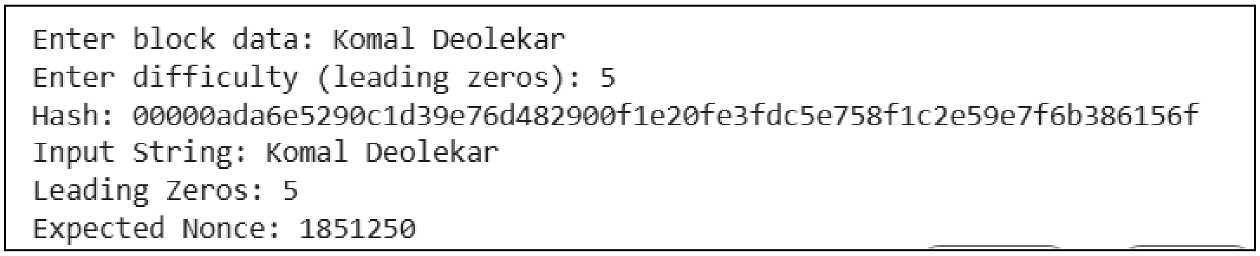


```
Enter a string: komal deolekar
Enter the nonce: 2
Hash Code: 7e57685a1e71427d7b4c29d1c15e4ab9027a5a59cbdf4d25c85c239e4705cfc9
```

Program #3: Solving Cryptographic Puzzle for Leading Zeros

```
import hashlib

def sha256_hash(data):
    # Calculate SHA-256 hash of the given data
    hash_object = hashlib.sha256(data.encode('utf-8'))
    return hash_object.hexdigest()

def hash_with_nonce(input_string, nonce):
    # Concatenate input string with nonce
    hash_string = input_string + str(nonce)
    # Generate hash for the combined string
    return sha256_hash(hash_string)

def find_nonce(input_string, num_zeros):
    # Initialize nonce value
    nonce = 0

    # Create required number of leading zeros
    hash_prefix = '0' * num_zeros

    while True:
        # Generate hash using input string and nonce
        hash_code = hash_with_nonce(input_string, nonce)

        # Check if hash has required leading zeros
        if hash_code.startswith(hash_prefix):
            print("Hash:", hash_code)
            return nonce

        # Increment nonce if condition is not satisfied
        nonce += 1

# Get user input
input_string = input("Enter block data: ")
num_zeros = int(input("Enter difficulty (leading zeros): "))

# Find the expected nonce
expected_nonce = find_nonce(input_string, num_zeros)

# Print results
print("Input String:", input_string)
print("Leading Zeros:", num_zeros)
print("Expected Nonce:", expected_nonce)
```

Output :



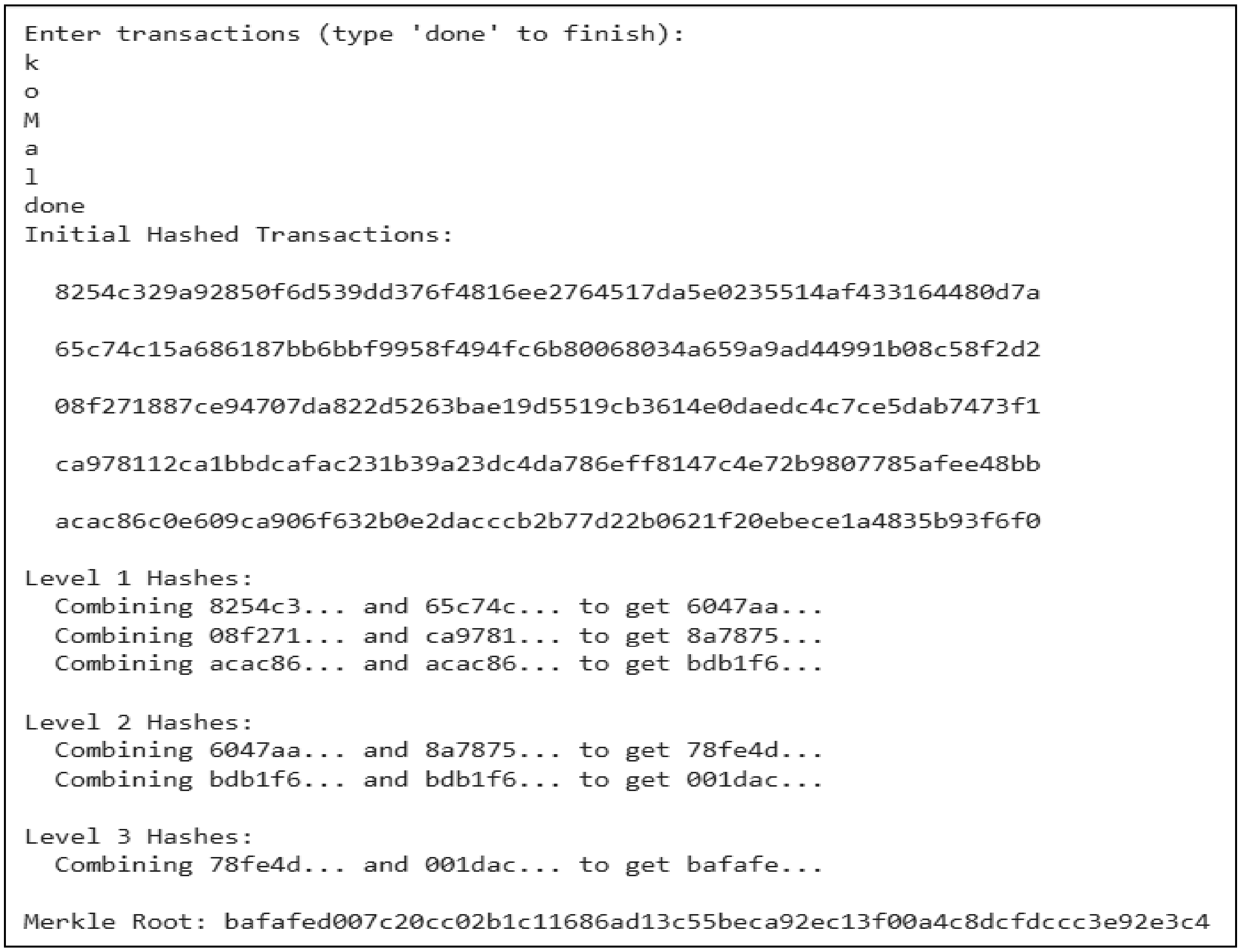
```
Enter block data: Komal Deolekar
Enter difficulty (leading zeros): 5
Hash: 00000ada6e5290c1d39e76d482900f1e20fe3fdc5e758f1c2e59e7f6b386156f
Input String: Komal Deolekar
Leading Zeros: 5
Expected Nonce: 1851250
```

Program #4: Generating Merkle Tree for a Given Set of Transactions

```python
import hashlib
def build_merkle_tree(transactions):
    # If no transactions are provided
    if len(transactions) == 0:
        return None
    # Hash each transaction initially
    current_level = [
        hashlib.sha256(tx.encode('utf-8')).hexdigest()
        for tx in transactions
    ]

    # Print initial transaction hashes
    print("Initial Hashed Transactions:",)
    for tx_hash in current_level:
        print(f"\n  {tx_hash}")

    level = 1

    # Build the Merkle Tree until one hash remains
    while len(current_level) > 1:
        print(f"\nLevel {level} Hashes:")

        # If number of hashes is odd, duplicate last hash
        if len(current_level) % 2 != 0:
            current_level.append(current_level[-1])
        new_level = []

        # Combine hashes pairwise
        for i in range(0, len(current_level), 2):
            combined = current_level[i] + current_level[i + 1]
            hash_combined = hashlib.sha256(combined.encode('utf-8')).hexdigest()
            new_level.append(hash_combined)
            # Print combining process (sample style)
            print(
                f"  Combining {current_level[i][:6]}... and "
                f"{current_level[i+1][:6]}... to get {hash_combined[:6]}..."
            )

        # Move to next level
        current_level = new_level
        level += 1

    # Return final Merkle Root
    return current_level[0]

# Get transactions from user
print("Enter transactions (type 'done' to finish):")
transactions = []

while True:
    tx = input()
    if tx.lower() == "done":
        break
    transactions.append(tx)

# Generate and print Merkle Root
if transactions:
    merkle_root = build_merkle_tree(transactions)
    print("\nMerkle Root:", merkle_root)
```

```
else:
    print("No transactions entered.")
```

Output :



```
Enter transactions (type 'done' to finish):
k
o
M
a
l
done
Initial Hashed Transactions:

  8254c329a92850f6d539dd376f4816ee2764517da5e0235514af433164480d7a

  65c74c15a686187bb6bbf9958f494fc6b80068034a659a9ad44991b08c58f2d2

  08f271887ce94707da822d5263bae19d5519cb3614e0daedc4c7ce5dab7473f1

  ca978112ca1bbdcafac231b39a23dc4da786eff8147c4e72b9807785afee48bb

  acac86c0e609ca906f632b0e2dacccb2b77d22b0621f20ebece1a4835b93f6f0

Level 1 Hashes:
  Combining 8254c3... and 65c74c... to get 6047aa...
  Combining 08f271... and ca9781... to get 8a7875...
  Combining acac86... and acac86... to get bdb1f6...

Level 2 Hashes:
  Combining 6047aa... and 8a7875... to get 78fe4d...
  Combining bdb1f6... and bdb1f6... to get 001dac...

Level 3 Hashes:
  Combining 78fe4d... and 001dac... to get bafafe...

Merkle Root: bafafed007c20cc02b1c11686ad13c55beca92ec13f00a4c8dcfdccc3e92e3c4
```

Conclusion:

In this experiment, cryptographic hashing using the SHA-256 algorithm was implemented to generate secure hash values for given data. The use of a nonce along with the input data demonstrated how small changes significantly affect the resulting hash. The Proof of Work mechanism was simulated by finding a nonce that satisfies a given difficulty condition, helping to understand the mining process in blockchain systems. The construction of a Merkle Tree showed how multiple transactions can be efficiently combined into a single Merkle Root, ensuring data integrity and quick verification. Overall, this experiment provided a clear understanding of the core mechanisms that ensure security, integrity, and reliability in blockchain technology.